Some Properties of Data Types with Inequations

Kondoh, Hidetaka

数理解析研究所講究録
第 851 巻 1993 年 1-20

Some Properties of Data Types with Inequations

Hidetaka Kondoh
Advanced Research Laboratory, Hitachi, Ltd.
Hatoyama, Saitama 350-03, Japan
kondoh@harl.hitach.co.jp

Abstract

This work aims to unify two approaches to abstract data types, one from logic (using typed λ-calculi) and the other from algebra (using first-order equational theory) by giving a domain-theoretic semantics. This paper presents a first-order type system of record types enriched with a set of inequations, as an approximated form of equational algebraic specification, to capture the notion of *structures*. We propose the notion of *algebraic types* by enriching the record type system with inequations and the notion of *algebraic inheritances*, an extension of the multiple inheritances à la Cardelli which incorporates the richness of structures, and show that our type system is a conservative extension of Cardelli's one by a purely syntactical way. Next we give a denotational semantics of our type system on the basis of the complete partial equivalence relation model on a cpo and this type system is shown to be sound with respect to this semantics. The extensions of the system to second-order calculi are remaining important themes and we give some considerations in the last section.

1. Introduction

There are essentially two approaches to formal modeling of abstract data types (*ADTs*): one is from logic, using typed λ-calculi; the other is from algebra, using first-order equational logic. But neither succeeds to fully capture our intuitions on abstract data types in computer programming. The final aim of our work is to integrate these two approaches and give a uniform semantics for all aspects of abstract data types principally using domain theory. This paper is the first step toward the goal, incorporating inequations as *assertions* (we reserve the term *axiom* for meta-theoretical usages) with the record type calculus à la Cardelli. It shows that our calculus supports a novel inheritance mechanism based on algebra-like structures of data types, conservatively extends the original calculus, and is sound with respect to the complete partial equivalence relation semantics of data types.

An ADT hides two kinds of information; one is the type of the representation of the data structure to be abstracted by that ADT, which we call the *representation type*; the other is a suite of implementations of *operations associated with* that ADT, which are well-typed with respect to the representation type chosen for that ADT, and we call the particular suite of types of associated operations for an ADT its *implementation type*. Then the inheritances between ADTs are the order relationships based on the relative "richness" of structures, e.g., stacks vs. dequeues, queues vs. dequeues, etc.

The algebraic approach models an ADT as a many-sorted first-order equational theory. Such theory is specified by a set of operator symbols (*signatures*) and equations to define the behavior of the associated operations (denoted by signatures) of the ADT [Ehrig and Mahr 85], but there is still disagreement about whether an ADT should be interpreted as the class of initial algebras or that of all algebras, etc. Furthermore, this approach has had little success in treating higher-order functions and in extending to higher-order logic systems for polymorphism, dependent types, etc.

The logical approach uses record types as the basic tool for modeling ADTs. Recent works on this approach can be classified into two streams; one is the modeling of inheritances in ADTs, introduced in Cardelli's pioneering paper, "A Semantics of Multiple Inheritances" [Cardelli 84]; the other focuses on the formalization of the information hiding mechanism of ADTs by existentially quantified types, a concept originated in "Abstract Data Types Have Existential Type" [Mitchell and Plotkin 85]. We summarize the correspondence between ADTs and existentially quantified record types according to the Cardelli-Mitchell-Plotkin modeling (Mitchell and Plotkin originally have used product types rather than record types, but we use record types as in [Cardelli and Wegner 85]).

abstract data type	existentially quantified record type
implementation type	record type
associated operator symbol	record field label
suite of associated operations	record value
associated operation	value bound to a record field label
inheritance relation	subtype relation on record types

Note that Cardelli's work and most works in his stream are on object-oriented programming rather than on ADTs. But Cardelli's idea to the method inheritances is applicable to inheritances of associated operations of ADTs by explicit parameterizations of methods with respect to objects' internal states, i.e., instance variables.

The essential difficulty in the above modeling is the ignorance of *algebraic structures* of ADTs in the sense of the algebraic approach. This problem has been pointed out by Reynolds in [Reynolds 83 and 85]. Thus the logical approach is far from satisfactory. In other words, only *anarchic* algebras correspond to record types, while the enriched record types of our system denote *non-anarchic* algebra-like structures, hence we call those enriched record types *algebraic types*.

The next section briefly introduces Cardelli's approach applied to inheritances among ADTs and point out the problem with the Cardelli-Mitchell-Plotkin modeling of ADTs. Section 3 proposes the notions of *algebraic type* and *algebraic inheritance*, and define a mini-language, μFinal, based on these ideas. Section 4 analyzes syntactical properties of the type system of μFinal. Section 5 gives the semantics of μFinal using partial equivalence relation models of types, and the first-order theory for typing rules of μFinal is shown to be sound with respect to this semantics. Finally, Section 6 summarizes relating this to other works and suggesting the direction of future works.

2. The Cardelli-Mitchell-Plotkin Approach and Its Problem

In this section we give an introduction to the Cardelli's work applied to the modeling of inheritances of ADTs and show the problem of the Cardelli-Mitchell-Plotkin modeling of ADTs.

$e \in \mathbf{Exp}$	The set of expressions:
$e ::= x$	(ordinary) variables,
$\mid c$	constants,
$\mid \lambda x{:}\sigma.e$	abstractions,
$\mid e_1 e_2$	applications,
$\mid \{l_1 = e_1, \ldots, l_n = e_n\}$	record expressions ($n \geq 0$),
$\mid e.l$	field selections,
$\mid [l = e]$	tagging expressions,
$\mid$ **case** e **of** l_1 **then** $e_1, \ldots, l_n$ **then** e_n	tag-case expressions ($n \geq 0$),
$\mid \mathbf{fix}(e)$	recursions by fixed-points.
$\sigma, \tau \in \mathbf{Type}$	The set of types:
$\sigma ::= \iota$	basic types,
$\mid \sigma_1 \to \sigma_2$	functional types,
$\mid \{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}$	record types ($n \geq 0$),
$\mid [l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n]$	variant types ($n \geq 0$).

Note: We leave details of the following syntactic categories unspecified:

$x \in \mathbf{Var}$	The set of variables;
$c \in \mathbf{Const}$	The set of constant symbols;
$l \in \mathbf{Label}$	The set of record field labels and variant tags;
$\iota \in \mathbf{BaseType}$	The finite set of base types (in Section 5 we assume $\mathbf{BaseType} = \{Bool, Nat\}$).

Figure 1. Syntax of μFun.

The syntax of the Cardelli's mini-language, which we hereafter call μFun, is given in Fig. 1. The order of field labels in record types and in record expressions is insignificant in the above production rules. Note that we the term "*expression*," rather than "*(pre-)term*" is used throughout this paper since it is more common in computer science.

Cardelli introduced the subtype relation of μFun on the basis of the inclusion relation between sets of field labels of record types. The judgment of the subtype relation is

$$\sigma <: \tau$$

and the relation is defined by the following axioms and rules:

$$\{\text{BASE}\} \quad \iota <: \iota$$

$$\{\text{TRANS}\} \quad \frac{\sigma_1 <: \sigma_2 \qquad \sigma_2 <: \sigma_3}{\sigma_1 <: \sigma_3}$$

$$\{\text{ARROW}\} \quad \frac{\sigma' <: \sigma \qquad \tau <: \tau'}{\sigma \to \tau \;<:\; \sigma' \to \tau'}$$

$$\{\text{RECORD}\} \quad \frac{\sigma_1 <: \tau_1 \;\ldots\; \sigma_n <: \tau_n}{\{l_1 : \sigma_1, \ldots, l_n : \sigma_n, \ldots, l_{n+m} : \sigma_{n+m}\} \;<:\; \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}$$

$$\{\text{VARIANT}\} \quad \frac{\sigma_1 <: \tau_1 \;\ldots\; \sigma_n <: \tau_n}{[l_1 : \sigma_1, \ldots, l_n : \sigma_n] \;<:\; [l_1 : \tau_1, \ldots, l_n : \tau_n, \ldots, l_{n+m} : \tau_{n+m}]}$$

Figure 2. The Subtyping Axioms and the Rules of μFun.

Here the rule {RECORD} is the essential rule in the modeling of inheritances as will be shown later.

The judgment of typing in μFun is of the form:

$$\Gamma \rhd e : \sigma$$

where Γ is a *basis*, i.e., a finite map from variables to types. We introduce notations for bases:

Notation 1.

(1) $\varnothing$ denotes the empty basis.

(2) Let Γ be a basis, x be a variable, and σ be a type. Then $\Gamma[x : \sigma]$ is the basis defined by the following finite map, Γ', such that for any variable y,

$$\Gamma'(y) = \begin{cases} \sigma, & (\text{if } x \equiv y) \\ \Gamma(y). & (\text{otherwise}) \end{cases}$$

(3) The notation, dom(Γ), denotes the set of variables on which the basis Γ is defined.

The typing axioms and rules of μFun are as follows, where FV(e) denotes the set of free variables in e:

$$[\text{VAR}]\quad \Gamma[x:\sigma] \rhd x:\sigma$$

$$[\text{CONST}]\quad \Gamma \rhd c_{ij} : \iota_i$$

$$[\text{WEAK}]\quad \frac{\Gamma \rhd e:\sigma}{\Gamma[x:\sigma'] \rhd e:\sigma} \qquad (x \notin \mathrm{FV}(e))$$

$$[\text{SUBTYPE}]\quad \frac{\Gamma \rhd e:\sigma \qquad \sigma <: \sigma'}{\Gamma \rhd e:\sigma'}$$

$$[\text{ABS}]\quad \frac{\Gamma[x:\sigma] \rhd e:\sigma'}{\Gamma \rhd (\lambda x{:}\sigma.e) : \sigma \to \sigma'}$$

$$[\text{APPL}]\quad \frac{\Gamma \rhd e:\sigma' \to \sigma \qquad \Gamma \rhd e':\sigma'}{\Gamma \rhd (ee') : \sigma}$$

$$[\text{RECORD}]\quad \frac{\Gamma \rhd e_1:\sigma_1 \ \ldots\ \Gamma \rhd e_n:\sigma_n}{\Gamma \rhd \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}}$$

$$[\text{SELECT}]\quad \frac{\Gamma \rhd e : \{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}}{\Gamma \rhd e.l_i : \sigma_i} \qquad (1 \le i \le n)$$

$$[\text{VARIANT}]\quad \frac{\Gamma \rhd e:\sigma}{\Gamma \rhd [l = e] : [l{:}\sigma]}$$

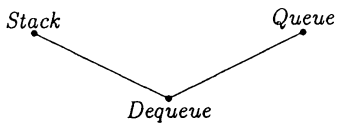
$$[\text{CASE}]\quad \frac{\Gamma \rhd e : [l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n] \qquad \Gamma \rhd e_1 : \sigma_1 \to \sigma \ \ldots\ \Gamma \rhd e_n : \sigma_n \to \sigma}{\Gamma \rhd (\textbf{case}\ e\ \textbf{of}\ l_1\ \textbf{then}\ e_1, \ldots, l_n\ \textbf{then}\ e_n) : \sigma}$$

$$[\text{FIX}]\quad \frac{\Gamma \rhd e : \sigma \to \sigma}{\Gamma \rhd \textbf{fix}(e) : \sigma}$$

Figure 3. The Typing Axioms and the Rules of μFun.

A concrete example demonstrates Cardelli's modeling of *multiple inheritances* in ADTs. To display examples compactly, we informally use Standard ML like syntax [MTH 90] for global definitions.

Example. *Stack* (of natural numbers) has as its equipped operations: *new*, to create a empty stack; *isnew*, to check a stack of its emptiness; *push*, to add some number to a stack; *top*, to see the the top (= lastly pushed) element; and *pop*, to remove the top element from a stack. On the other hand, *Queue* is characterized by the following operations: *new*, to make a empty queue; *isnew*, to check emptiness of a queue; *add*, to add a number at the end of a queue; *first*, to get the head (= the firstly added) element; and *remove*, to discard the head element from a queue. And *Dequeue* is equipped with all of the operations of both *Stack* and *Queue*. Then the inheritance hierarchy of these types is as in the following diagram:



Suppose we have *List* as a standard type constructor, and *Nat* and *Bool* as base types in μFun, and we select the list of natural numbers as the common representation types for these ADTs, i.e.:

type *StackValRep* = *List*[*Nat*];
type *QueueValRep* = *List*[*Nat*];
type *DequeueValRep* = *List*[*Nat*];

Then we can define their implementation types:

$$\begin{aligned}\textbf{type}\ StackOpImpl = \{&new\colon StackValRep,\\ &isnew\colon StackValRep \to Bool,\\ &push\colon Nat \to StackValRep \to StackValRep,\\ &top\colon StackValRep \to Nat,\\ &pop\colon StackValRep \to StackValRep\};\end{aligned}$$

$$\begin{aligned}\textbf{type}\ QueueOpImpl = \{&new\colon QueueValRep,\\ &isnew\colon QueueValRep \to Bool,\\ &add\colon Nat \to QueueValRep \to QueueValRep,\\ &first\colon QueueValRep \to Nat,\\ &remove\colon QueueValRep \to QueueValRep\};\end{aligned}$$

$$\begin{aligned}\textbf{type}\ DequeueOpImpl = \{&new\colon DequeueValRep,\\ &isnew\colon DequeueValRep \to Bool,\\ &push\colon Nat \to DequeueValRep \to DequeueValRep,\\ &top\colon DequeueValRep \to Nat,\\ &pop\colon DequeueValRep \to DequeueValRep,\\ &add\colon Nat \to DequeueValRep \to DequeueValRep,\\ &first\colon DequeueValRep \to Nat,\\ &remove\colon DequeueValRep \to DequeueValRep\};\end{aligned}$$

Now we can give a suite of implementations of equipped operations of the type *Stack* as a record expression as follows:

$$\begin{aligned}\textbf{val}\ aStackOpImpl = \{&new = nil,\\ &isnew = \lambda s\colon StackValRep.isnull(s),\\ &push = \lambda i\colon Nat.\lambda s\colon StackValRep.cons(i)(s),\\ &pop = \lambda s\colon StackValRep.tail(s),\\ &top = \lambda s\colon StackValRep.head(s)\};\end{aligned}$$

This behaves in the *last-in first-out* manner as expected for stacks. By this mechanism, for example,

$$\begin{aligned}&aStackOpImpl.top(\\ &\quad aStackOpImpl.pop(aStackOpImpl.push(2)(aStackOpImpl.push(1)(aStackOpImpl.new))))\end{aligned}$$

yields 1. On the other hand, with the following suite of implementations

$$\begin{aligned}\textbf{val}\ anotherStackOpImpl = \{&new = nil,\\ &isnew = \lambda s\colon StackValRep.isnull(s),\\ &push = \lambda i\colon Nat.\lambda s\colon StackValRep.cons(i)(s),\\ &pop = \textbf{fix}(\lambda p\colon StackValRep \to StackValRep.\lambda s\colon StackValRep.\\ &\qquad \textbf{if}\ length(s) \leq 1\ \textbf{then}\ nil\ \textbf{else}\ cons(head(s))(p(tail(s)))),\\ &top = \textbf{fix}(\lambda t\colon StackValRep \to Nat.\lambda s\colon StackValRep.\\ &\qquad \textbf{if}\ length(s) \leq 1\ \textbf{then}\ head(s)\ \textbf{else}\ t(tail(s)))\};\end{aligned}$$

where *length* is the usual length function for lists, the value of the expression

$$\begin{aligned}&anotherStackOpImpl.top(anotherStackOpImpl.pop(\\ &\quad anotherStackOpImpl.push(2)(anotherStackOpImpl.push(1)(anotherStackOpImpl.new))))\end{aligned}$$

is 2, since the *anotherStackOpImpl* acts in the *first-in first-out* fashion. In fact, *anotherStackOpImpl* is an implementation suite adequate for queues rather than for stacks but still has the type *StackOpImpl*.

From this example, we can see that the Cardelli-Mitchell-Plotkin modeling cannot distinguish between behaviors of stacks and of queues, and treats identically stacks and queues having the same type. This limitation of their approach is the problem which this work attempts to solve.

3. Algebraic Types and Algebraic Inheritances

In the last section, we saw that record types cannot capture all of the aspects of the implementation types of abstract data types. In order to overcome this difficulty, we extend the type system of μFun with *inequational assertions* for record types and construct a new language μFinal (μ*F*un with *In*heritances between *Al*gebraic types). We call such augmented record types *algebraic types* because of the analogy to algebras with equational specifications.

The syntax of μFinal is an extension of that of μFun with the following production rules.

$e \in \mathbf{Exp}$	The set of expressions:
$e ::= r$	implementation variables.
$\sigma, \tau \in \mathbf{Type}$	The set of types:
$\sigma ::= \rho r{:}\tau.\{\phi_1, \ldots, \phi_k\}$	algebraic types (τ is a record type, $k \geq 0$).
$\phi, \psi \in \mathbf{Assertion}$	The set of assertions:
$\phi ::= e_1 \leqslant e_2 : \sigma$	atomic assertions,
$\mid$ **forall** $x{:}\sigma.\phi$	quantified assertions.

Note: We leave details of the following syntactic category unspecified:

$r \in \mathbf{IVar}$ The set of implementation variables.

Figure 4. The Characteristic Syntax Rules of μFinal.

In the above rules, the order of occurrences of assertions in an algebraic type is insignificant as is the case for the order of field labels, and the change of implementation variables in ρ-binding is also insignificant like in the ususal λ-binding. Before stating syntactical constraints to μFinal, we need a definition, which is analogous to the notion of *active subexpression* in [Plotkin 77].

Definition 2. Let e, e' be expressions of μFinal. Then e' *strictly occurs in* e iff one of the following conditions holds:

(1) $e \equiv x$ and $e' \equiv x$,
(2) $e \equiv r$ and $e' \equiv r$,
(3) $e \equiv e''.l$ and e' strictly occurs in e'',
(4) $e \equiv e''e'''$ and e' strictly occurs in e'',
(5) $e \equiv$ **case** e'' l_1 **then** $e_1, \ldots, l_n$ **then** e_n and e' strictly occurs in e'', or
(6) $e \equiv \mathbf{fix}(e'')$ and e' strictly occurs in e''.

Then the syntactical constraints to μFinal are:

(a) each assertion of an algebraic type must be closed by **forall** quantification except for free occurrences of the implementation variable bound by the algebraic type containing that assertion; and
(b) the implementation variable of an algebraic type must strictly occur in the left-hand expression of each assertion of the algebraic type.

The first constraint is necessary for giving semantics for the proof theory of μFinal, while the second one is essential for constructing semantics of μFinal.

Notation 3. We identify each record type with an algebraic type with null assertion, e.g.

$$\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\} \equiv \rho r{:}\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}.\{\};$$

and we sometimes write algebraic types in more intuitive form, e.g.

$$\rho r.\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n \mid \phi_1, \ldots, \phi_k\} \stackrel{\text{abbrev}}{=} \rho r{:}\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}.\{\phi_1, \ldots, \phi_k\}.$$

We often abbreviate symmetrical pairs of inequations as an equation, e.g.

$$\textbf{forall}\ x_1{:}\sigma_1.\cdots.\textbf{forall}\ x_n{:}\sigma_n.e = e' : \tau \stackrel{\text{abbrev}}{=}$$
$$\textbf{forall}\ x_1{:}\sigma_1.\cdots.\textbf{forall}\ x_n{:}\sigma_n.e \leqslant e' : \tau,\ \textbf{forall}\ x_1{:}\sigma_1.\cdots.\textbf{forall}\ x_n{:}\sigma_n.e' \leqslant e : \tau.$$

We also omit implementation variables in assertions and write l for $r.l$ when there is no danger of confusion.

For the type system of μFinal, we introduce a first-order theory of μFinal.

Definition 4.

(1) μFINAL is the first-order theory with axioms and rules which will be described in this section and with the following three forms of judgments as sentences:

- $\sigma <: \tau$ for subtyping,
- $\Gamma, \Delta \triangleright e{:}\sigma$ for typing,
- $\Gamma, \Delta \triangleright \phi$ for assertions,

where Δ is a basis for implementation variables. The deducibility in μFINAL is shown by $\vdash_{\mu\text{FINAL}}$.

(2) For the type system of μFun, μFUN is defined in the same way, and $\vdash_{\mu\text{FUN}}$ denotes its deducibility.

Note: We usually omit the subscripts and simply write $\vdash$ when there is no danger of confusion.

First we define the subtype relation on μFinal. The {RECORD} rule of μFun is generalized to handle assertions.

$$\{\text{ALGEBRA}\}\ \frac{\bigwedge_{i=1}^{k}(\varnothing, \{r : \sigma\} \triangleright \phi_i[r_1 := r]) \vdash_{\mu\text{FINAL}} \bigwedge_{j=1}^{l}(\varnothing, \{r : \tau\} \triangleright \psi_j[r_2 := r]) \qquad \sigma_1 <: \tau_1\ \ldots\ \sigma_n <: \tau_n}{\rho r_1{:}\{l_1{:}\sigma_1, \ldots, l_{n+m}{:}\sigma_{n+m}\}.\{\phi_1, \ldots, \phi_k\}\ <:\ \rho r_2{:}\{l_1{:}\tau_1, \ldots, l_n{:}\tau_n\}.\{\psi_1, \ldots, \psi_l\}} \quad (m \geq 0)$$

where

$\bigwedge_{i=1}^{k}(\varnothing, \{r : \sigma\} \triangleright \phi_i[r_1 := r]) \vdash_{\mu\text{FINAL}} \bigwedge_{j=1}^{l}(\varnothing, \{r : \tau\} \triangleright \psi_j[r_2 := r])$ is a short-hand notation meaning that for each $1 \leq j \leq l$,

$$\varnothing, \{r : \sigma\} \triangleright \phi_1[r_1 := r], \ldots, \varnothing, \{r : \sigma\} \triangleright \phi_k[r_1 := r] \vdash_{\mu\text{FINAL}} \varnothing, \{r : \tau\} \triangleright \psi_j[r_2 := r];$$

r is a fresh implementation variable;

$\sigma \equiv \{l_1{:}\sigma_1, \ldots, l_{n+m}{:}\sigma_{n+m}\}$;

$\tau \equiv \{l_1{:}\tau_1, \ldots, l_n{:}\tau_n\}$.

Figure 5. The Characteristic Subtyping Rule of μFinal.

Intuitively speaking, this {ALGEBRA} rule states that if an algebraic type is a subtype of another one in the sense of record types (i.e. $\sigma <: \tau$) and the set of assertions of the subtype, $\{\phi_1, \ldots, \phi_k\}$, is stronger than that of the other, $\{\psi_1, \ldots, \psi_l\}$, then it is a subtype of the other as algebraic types. Owing to this rule, the subtype relation in μFinal becomes a preorder but not a partial order as in the case of μFun. We call the multiple inheritances based on this subtype relation *algebraic inheritances*, since they reflect the richness of algebra-like structures of data types.

For typing in μFinal, we replace each judgment of the form $\Gamma \triangleright e : \sigma$ in the typing axioms and rules of μFun by one of the form $\Gamma, \Delta \triangleright e : \sigma$ augmented with a basis for implementation variables. Furthermore, we have to add an axiom and two rules:

$$[\text{IVAR}]\quad \Gamma, \Delta[r : \tau] \triangleright r : \tau$$

$$[\text{IWEAK}]\quad \frac{\Gamma, \Delta \triangleright e : \sigma}{\Gamma, \Delta[r : \tau] \triangleright e : \sigma} \qquad (r \notin \text{FV}(e))$$

$$[\text{EXTEND}]\quad \frac{\Gamma, \Delta \triangleright e : \rho r{:}\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}.\{\phi_1, \ldots, \phi_k\} \qquad \Gamma, \Delta \triangleright \phi_{k+1}[r := e]}{\Gamma, \Delta \triangleright e : \rho r{:}\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}.\{\phi_1, \ldots, \phi_{k+1}\}}$$

Figure 6. The Characteristic Typing Axiom and the Rules of μFinal.

The [EXTEND] rule means that if the expression e satisfies the assertion ϕ_{k+1}, then we can add this assertion to the algebraic type of e.

$$\langle \text{VAR} \rangle \quad \Gamma[x:\sigma], \Delta \rhd x \leqslant x : \sigma$$

$$\langle \text{IVAR} \rangle \quad \Gamma, \Delta[r:\tau] \rhd r \leqslant r : \tau$$

$$\langle \text{CONST} \rangle \quad \Gamma, \Delta \rhd c_{ij} \leqslant c_{ij} : \iota_i$$

$$\langle \text{TRANS} \rangle \frac{\Gamma, \Delta \rhd e_1 \leqslant e_2 : \sigma \qquad \Gamma, \Delta \rhd e_2 \leqslant e_3 : \sigma}{\Gamma, \Delta \rhd e_1 \leqslant e_3 : \sigma}$$

$$\langle \text{WEAK} \rangle \frac{\Gamma, \Delta \rhd e_1 \leqslant e_2 : \sigma}{\Gamma[x:\sigma'], \Delta \rhd e_1 \leqslant e_2 : \sigma} \qquad (x \notin \text{FV}(e_1) \cup \text{FV}(e_2))$$

$$\langle \text{IWEAK} \rangle \frac{\Gamma, \Delta \rhd e_1 \leqslant e_2 : \sigma}{\Gamma, \Delta[r:\tau] \rhd e_1 \leqslant e_2 : \sigma} \qquad (r \notin \text{FV}(e_1) \cup \text{FV}(e_2))$$

$$\langle \text{SUBTYPE} \rangle \frac{\Gamma, \Delta \rhd e_1 \leqslant e_2 : \sigma \qquad \sigma <: \sigma'}{\Gamma, \Delta \rhd e_1 \leqslant e_2 : \sigma'}$$

$$\langle \text{forall-E} \rangle \frac{\Gamma, \Delta \rhd \textbf{forall}\ x{:}\sigma.\phi \qquad \Gamma, \Delta \rhd e : \sigma}{\Gamma, \Delta \rhd \phi[x := e]}$$

$$\langle \text{forall-I} \rangle \frac{\Gamma[x:\sigma], \Delta \rhd \phi}{\Gamma, \Delta \rhd \textbf{forall}\ x{:}\sigma.\phi} \qquad (x \notin \text{dom}(\Gamma))$$

$$\langle \beta_{\text{func}} \rangle \frac{\Gamma[x:\sigma], \Delta \rhd e : \sigma' \qquad \Gamma, \Delta \rhd e' : \sigma}{\Gamma, \Delta \rhd (\lambda x{:}\sigma.e)e' = e[x := e'] : \sigma'}$$

$$\langle \text{ABS} \rangle \frac{\Gamma[x:\sigma], \Delta \rhd e \leqslant e' : \sigma'}{\Gamma, \Delta \rhd (\lambda x{:}\sigma.e) \leqslant (\lambda x{:}\sigma.e') : \sigma \to \sigma'}$$

$$\langle \text{APPL} \rangle \frac{\Gamma, \Delta \rhd e_1 \leqslant e_1' : \sigma' \to \sigma \qquad \Gamma, \Delta \rhd e_2 \leqslant e_2' : \sigma'}{\Gamma, \Delta \rhd (e_1 e_2) \leqslant (e_1' e_2') : \sigma}$$

$$\langle \beta_{\text{record}} \rangle \frac{\Gamma, \Delta \rhd e_1 : \sigma_1 \ \ldots\ \Gamma, \Delta \rhd e_n : \sigma_n}{\Gamma, \Delta \rhd \{l_1 = e_1, \ldots, l_n = e_n\}.l_i = e_i : \sigma_i} \qquad (1 \le i \le n)$$

$$\langle \text{RECORD} \rangle \frac{\Gamma, \Delta \rhd e_1 \leqslant e_1' : \sigma_1 \ \ldots\ \Gamma, \Delta \rhd e_n \leqslant e_n' : \sigma_n}{\Gamma, \Delta \rhd \{l_1 = e_1, \ldots, l_n = e_n\} \leqslant \{l_1 = e_1', \ldots, l_n = e_n'\} : \{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}}$$

$$\langle \text{SELECT} \rangle \frac{\Gamma, \Delta \rhd e \leqslant e' : \{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}}{\Gamma, \Delta \rhd e.l_i \leqslant e'.l_i : \sigma_i} \qquad (1 \le i \le n)$$

$$\langle \text{ASSERT} \rangle \frac{\Gamma, \Delta \rhd e : \rho r{:}\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}.\{\phi_1, \ldots, \phi_k\}}{\Gamma, \Delta \rhd \phi_i[r := e]} \qquad (1 \le i \le k)$$

$$\langle \beta_{\text{variant}} \rangle \frac{\Gamma, \Delta \rhd e : \sigma_i \qquad \Gamma, \Delta \rhd e_1 : \sigma_1 \to \sigma' \ \ldots\ \Gamma, \Delta \rhd e_n : \sigma_n \to \sigma'}{\Gamma, \Delta \rhd (\textbf{case}\ [l_i = e]\ \textbf{of}\ l_1\ \textbf{then}\ e_1, \ldots, l_n\ \textbf{then}\ e_n) = (e_i e) : \sigma'} \qquad (1 \le i \le n)$$

$$\langle \text{VARIANT} \rangle \frac{\Gamma, \Delta \rhd e \leqslant e' : \sigma}{\Gamma, \Delta \rhd [l = e] \leqslant [l = e'] : [l{:}\sigma]}$$

$$\langle \text{CASE} \rangle \frac{\Gamma, \Delta \rhd e \leqslant e' : [l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n] \qquad \Gamma, \Delta \rhd e_1 \leqslant e_1' : \sigma_1 \to \sigma' \ \ldots\ \Gamma, \Delta \rhd e_n \leqslant e_n' : \sigma_n \to \sigma'}{\Gamma, \Delta \rhd (\textbf{case}\ e\ \textbf{of}\ l_1\ \textbf{then}\ e_1, \ldots, l_n\ \textbf{then}\ e_n) \leqslant (\textbf{case}\ e'\ \textbf{of}\ l_1\ \textbf{then}\ e_1', \ldots, l_n\ \textbf{then}\ e_n') : \sigma'}$$

$$\langle \beta_{\text{fix}} \rangle \frac{\Gamma, \Delta \rhd e : \sigma \to \sigma}{\Gamma, \Delta \rhd \textbf{fix}(e) = e(\textbf{fix}(e)) : \sigma}$$

$$\langle \text{FIX} \rangle \frac{\Gamma, \Delta \rhd e \leqslant e' : \sigma \to \sigma}{\Gamma, \Delta \rhd \textbf{fix}(e) \leqslant \textbf{fix}(e') : \sigma}$$

Figure 7. The Axioms and the Rules for Assertions of μFinal.

Finally, we must give the rules for inferring assertions. These rules are shown in Fig. 7. Note that rules, $\langle\beta_{\text{func}}\rangle$, $\langle\beta_{\text{record}}\rangle$, $\langle\beta_{\text{variant}}\rangle$ and $\langle\beta_{\text{fix}}\rangle$, whose conclusions have an equational form, actually denote a pair of rules each by the notational convention as stated before, and the meaning of each rule is apparent except for $\langle$ASSERT$\rangle$. This rule states that if an expression e has an algebraic type with assertions $\phi_1, \ldots, \phi_k$ in which the representation variable r denotes e, then we can use an instance of each assertion ϕ_i by substituting that expression e for r.

We end this section by giving an example of algebraic inheritances. We can define the implementation type of *stack* in the last section as an algebraic type.

type $StackOpImpl = \rho aStackOpImpl.\{new\colon StackValRep,$
$\quad isnew\colon StackValRep \to Bool,$
$\quad push\colon Nat \to StackValRep \to StackValRep,$
$\quad top\colon StackValRep \to Nat,$
$\quad pop\colon StackValRep \to StackValRep$
(*push-pop*) $\quad \mid$ **forall** $i\colon Nat$.**forall** $s\colon StackValRep.$
$\qquad pop(push(i)(s)) \leqslant s : StackValRep,$
$\quad$ **forall** $i\colon Nat$.**forall** $s\colon StackValRep.top(push(i)(s)) \leqslant i : Nat,$
$\quad$ **forall** $i\colon Nat$.**forall** $s\colon StackValRep.isnew(push(i)(s)) \leqslant false : Bool,$
$\quad isnew(new) \leqslant true : Bool\};$

Now consider another type *StackoidOpImpl*:

type $StackoidOpImpl = \rho aStackoidOpImpl.\{new\colon StackValRep,$
$\quad isnew\colon StackValRep \to Bool,$
$\quad push\colon Nat \to StackValRep \to StackValRep,$
$\quad top\colon StackValRep \to Nat,$
$\quad pop\colon StackValRep \to StackValRep$
(*push^2-pop^2*) $\quad \mid$ **forall** $i\colon Nat$.**forall** $j\colon Nat$.**forall** $s\colon StackValRep.$
$\qquad pop(pop(push(i)(push(j)(s)))) \leqslant s : StackValRep,$
$\quad$ **forall** $i\colon Nat$.**forall** $s\colon StackValRep.top(push(i)(s)) \leqslant i : Nat,$
$\quad$ **forall** $i\colon Nat$.**forall** $s\colon StackValRep.$
$\qquad isnew(push(i)(s)) \leqslant false : Bool,$
$\quad isnew(new) \leqslant true : Bool\};$

Clearly the assertion (*push-pop*) in *StackOpImpl* is stronger than (*push^2-pop^2*) in *StackoidOpImpl*; hence, any stack can be used as a stackoid. Therefore, *StackOpImpl* inherits the structure of *StackoidOpImpl*, and this fact is expressed in μFinal as the subtype relationship: $StackOpImpl <: StackoidOpImpl$.

Compare the definition of the type *QueueOpImpl* with the same signature as *StackOpImpl*:

type $QueueOpImpl = \rho aQueueOpImpl.\{new\colon StackValRep,$
$\quad isnew\colon StackValRep \to Bool,$
$\quad push\colon Nat \to StackValRep \to StackValRep,$
$\quad top\colon StackValRep \to Nat,$
$\quad pop\colon StackValRep \to StackValRep$
$\quad \mid$ **forall** $i\colon Nat$.**forall** $s\colon StackValRep.$
$\qquad pop(push(i)(s)) \leqslant$ **if** $isnew(s)$ **then** s
$\qquad\qquad$ **else** $push(i)(pop(s)) : StackValRep,$
$\quad$ **forall** $i\colon Nat$.**forall** $s\colon StackValRep.$
$\qquad top(push(i)(s)) \leqslant$ **if** $isnew(s)$ **then** i **else** $top(s) : Nat,$
$\quad$ **forall** $i\colon Nat$.**forall** $s\colon StackValRep.isnew(push(i)(s)) \leqslant false : Bool,$
$\quad isnew(new) \leqslant true : Bool\};$

Then clearly $StackOpImpl \not<: QueueOpImpl$ and $QueueOpImpl \not<: StackOpImpl$. Moreover, for *aStackOpImpl* and *anotherStackOpImpl* in Section 2, we can show $\vdash_{\mu\text{FINAL}} \varnothing, \varnothing \rhd aStackOpImpl : StackOpImpl$ and $\vdash_{\mu\text{FINAL}} \varnothing, \varnothing \rhd anotherStackOpImpl : QueueOpImpl$ as we have pointed out in Section 2 (assuming assertions on list operations are given).

4. Proof Theoretical Investigations of Algebraic Types

In this section we investigate the proof theoretical properties of the type system μFINAL. Especially we show the system is a conservative extension of μFUN.

First, we define classes of types, expressions, bases and sentences of μFinal having correspondences in μFun.

Definition 5.

(1) A type, σ, of μFinal is said to be *assertion-free* iff one of the following conditions holds:
   (a) $\sigma \equiv \iota$;
   (b) $\sigma \equiv \sigma_1 \to \sigma_2$, where each σ_i $(i = 1, 2)$ is assertion-free;
   (c) $\sigma \equiv \{l_1 : \sigma_1, \ldots, l_n : \sigma_n\}$, where each σ_i $(1 \leq i \leq n)$ is assertion-free; or
   (d) $\sigma \equiv [l_1 : \sigma_1, \ldots, l_n : \sigma_n]$, where each σ_i $(1 \leq i \leq n)$ is assertion-free.

(2) An expression, e, of μFinal is said to be *assertion-free* iff both of the following conditions hold:
   (a) e does not contain any implementation variable; and
   (b) each λ-abstraction occurring in e binds a variable with an assertion-free type.

(3) A basis, Γ, is *assertion-free* iff Γ assigns assertion-free types to each variable.

(4) A sentence, Σ, of μFINAL is *assertion-free* iff Σ has either one of the following forms:
   (a) $\Sigma \equiv \Gamma, \varnothing \rhd e : \sigma$, where Γ, e and σ are all assertion-free; or
   (b) $\Sigma \equiv \sigma <: \tau$, where σ and τ are assertion-free.

For each assertion-free sentence of μFINAL, we define the correspondence in μFUN.

Definition 6. Let Σ be an assertion-free sentence of μFINAL. Then its *corresponding sentence in* μFUN (notation: $\mu\text{FUN}(\Sigma)$) is defined as follows:

(1) when $\Sigma \equiv \Gamma, \varnothing \rhd e : \sigma$, $\mu\text{FUN}(\Sigma) \equiv \Gamma \rhd e : \sigma$;
(2) when $\Sigma \equiv \sigma <: \tau$, $\mu\text{FUN}(\Sigma) \equiv \Sigma$.

Next we define a function which removes all assertions from types.

Definition 7. The function aft : **Type** $\to$ **Type** is defined such that

(1) $\text{aft}(\iota) = \iota$,
(2) $\text{aft}(\sigma \to \tau) = \text{aft}(\sigma) \to \text{aft}(\tau)$,
(3) $\text{aft}(\{l_1 : \sigma_1, \ldots, l_n : \sigma_n\}) = \{l_1 : \text{aft}(\sigma_1), \ldots, l_n : \text{aft}(\sigma_n)\}$,
(4) $\text{aft}([l_1 : \sigma_1, \ldots, l_n : \sigma_n]) = [l_1 : \text{aft}(\sigma_1), \ldots, l_n : \text{aft}(\sigma_n)]$,
(5) $\text{aft}(\rho r : \tau.\{\phi_1, \ldots, \phi_k\}) = \text{aft}(\tau)$.

This aft is extended on **Exp**, **Assertion**, bases and sentences of μFINAL in the obvious way.

Then the following lemmas clearly hold:

Lemma 8. *Let τ be a type of* μFinal. *Then*
(1) $\text{aft}(\tau)$ *is assertion-free; and*
(2) if τ is assertion-free, then $\text{aft}(\tau) = \tau$. ∎

Lemma 9. *Let σ and τ be types of* μFinal *such that $\sigma <: \tau$. Then either one of the following cases holds:*
(1) $\sigma \equiv \tau \equiv \iota$;
(2) $\sigma \equiv \sigma_1 \to \sigma_2$ and $\tau \equiv \tau_1 \to \tau_2$, where $\tau_1 <: \sigma_1$ and $\sigma_2 <: \tau_2$;
(3) $\sigma \equiv \{l_1 : \sigma_1, \ldots, l_n : \sigma_n\}$ and $\tau \equiv \{l_1 : \tau_1, \ldots, l_m : \tau_m\}$, where $m \leq n$ and $\sigma_i <: \tau_i$ for all $1 \leq i \leq n$;
(4) $\sigma \equiv [l_1 : \sigma_1, \ldots, l_n : \sigma_n]$ and $\tau \equiv [l_1 : \tau_1, \ldots, l_m : \tau_m]$, where $m \geq n$ and $\sigma_i <: \tau_i$ for all $1 \leq i \leq m$; or
(5) $\sigma \equiv \rho r : \sigma'.\{\phi_1, \ldots, \phi_k\}$ and $\tau \equiv \rho r' : \tau'.\{\psi_1, \ldots, \psi_l\}$, where $\sigma' <: \tau'$. ∎

Lemma 10.
(1) If $x \in \mathbf{Var}$ *is not free in* e, *then*

$$\vdash \Gamma[x:\tau], \Delta \rhd e:\sigma \quad \Longrightarrow \quad \vdash \Gamma, \Delta \rhd e:\sigma.$$

(2) If $r \in \mathbf{IVar}$ *is not free in* e, *then*

$$\vdash \Gamma, \Delta[r:\tau] \rhd e:\sigma \quad \Longrightarrow \quad \vdash \Gamma, \Delta \rhd e:\sigma.$$

Proof. (1), (2) By induction on the number of steps of the proof. ∎

Lemma 11. *Let* σ *and* τ *be assertion-free types of* μFinal. *Then in* μFINAL

$$\vdash \sigma <: \tau \quad \Longrightarrow \quad \vdash \mathrm{aft}(\sigma) <: \mathrm{aft}(\tau).$$

Proof. By induction on the structure of σ using Lemma 9. ∎

Lemma 12. *Let* Γ *and* e *be assertion-free. Then*

$$\vdash \Gamma, \Delta \rhd e:\sigma \quad \Longrightarrow \quad \vdash \Gamma, \varnothing \rhd e:\mathrm{aft}(\sigma).$$

Proof. By induction on the number of steps of the proof of $\Gamma, \Delta \rhd e:\sigma$, we have to show that $\vdash \Gamma, \Delta \rhd e:\mathrm{aft}(\sigma)$ holds. Then we can obtain the desired result by repeated applications of Lemma 10. On induction, we show only non-trivial cases, i.e., the last step of the proof is either [SUBTYPE] or [EXTEND]. Other cases are obvious, since, in other rules, each expression in the premise is a subexpression of the expression in the conclusion.

Case 1: the [SUBTYPE] rule. $\Gamma, \Delta \rhd e:\sigma$ is deduced from $\Gamma, \Delta \rhd e:\sigma'$ and $\sigma' <: \sigma$. Hence $\Gamma, \Delta \rhd e:\mathrm{aft}(\sigma')$ by the induction hypothesis and $\mathrm{aft}(\sigma') <: \mathrm{aft}(\sigma)$ by Lemma 11. Therefore, by applying [SUBTYPE], we obtain $\Gamma, \Delta \rhd e:\mathrm{aft}(\sigma)$.

Case 2: the [EXTEND] rule. From the assumption, σ has the form $\rho r{:}\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}.\{\phi_1, \ldots, \phi_{k+1}\}$ and $\Gamma, \Delta \rhd e:\sigma$ is deduced from $\Gamma, \Delta \rhd e:\rho r{:}\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}.\{\phi_1, \ldots, \phi_k\}$ and $\Gamma, \Delta \rhd \phi_{k+1}[r := e]$. Hence we conclude $\Gamma, \Delta \rhd e:\sigma'$, where

$$\begin{aligned}\sigma' &\stackrel{\mathrm{def}}{=} \mathrm{aft}(\rho r{:}\{l_1{:}\sigma_1, \ldots, l_n{:}\sigma_n\}.\{\phi_1, \ldots, \phi_k\}) \\ &\equiv \{l_1{:}\mathrm{aft}(\sigma_1), \ldots, l_n{:}\mathrm{aft}(\sigma_n)\} \\ &\equiv \mathrm{aft}(\sigma). \ ∎\end{aligned}$$

Finally we can show the desired result.

Theorem 13. *If an assertion-free sentence* Σ *is provable in* μFINAL, *then there is a proof of* Σ *comprising only assertion-free sentences.*

Proof. To prove this, we have to check that if the conclusion is assertion-free then any premises are also assertion-free and do not need types of implementation variables for each rule. This is obvious for most rules; we only need to show it for {TRANS}, [SUBTYPE] and [APPL]. (Note that under the assumption of the assertion-freeness of the conclusion, {ALGEBRA} becomes the same as {RECORD} of μFUN, and [IVAR] and [EXTEND] are excluded.)

Case 1: the {TRANS} rule. By Lemma 9 and induction on the structure of types.

Case 2: the [SUBTYPE] rule. We must show that for any assertion-free Γ, Δ, e and σ', there is some assertion-free σ'' such that $\Gamma, \varnothing \rhd e:\sigma''$ and $\sigma'' <: \sigma'$, under the assumption that $\Gamma, \Delta \rhd e:\sigma$ and $\sigma <: \sigma'$ for some (possibly non-assertion-free) type σ.

Applying Lemmas 11, 8, and 12 to the assumption, we obtain $\Gamma, \varnothing \rhd e:\mathrm{aft}(\sigma)$ and $\mathrm{aft}(\sigma) <: \sigma'$. Therefore we can choose $\mathrm{aft}(\sigma)$ as the assertion-free σ''.

Case 3: the [APPL] rule. Similar to the above case using Lemma 12. ∎

As stated before, any assertion-free sentence Σ of μFINAL corresponds to some sentence of μFUN.

Corollary 14 Conservative Extension Theorem. *The theory* μFINAL *is a conservative extension of* μFUN. *That is, for any assertion-free sentence* Σ *of* μFINAL,

$$\vdash_{\mu\text{FINAL}} \Sigma \iff \vdash_{\mu\text{FUN}} \mu\text{FUN}(\Sigma). \blacksquare$$

The type system of μFinal is clearly undecidable since it has a power to specify a kind of partial correctness of functional programs. But the system restores decidability by forgetting all assertions, hence our system μFINAL can be viewed as a type system for specification/verification while μFUN is its decidable subsystem for compile-time type-checking. Then the following theorem states that "*all correct programs pass compilers.*"

Theorem 15. *For any* Γ, Δ, σ, *and any* e *without implementation variables,*

$$\vdash_{\mu\text{FINAL}} \Gamma, \Delta \rhd e : \sigma \quad \Longrightarrow \quad \vdash_{\mu\text{FUN}} \text{aft}(\Gamma) \rhd \text{aft}(e) : \text{aft}(\sigma).$$

Proof. Similar to the proof of Theorem 13. ∎

5. Semantics of Algebraic Types and Algebraic Inheritances

In this section, we give a denotational semantics of μFinal and show that the theory μFINAL in Section 4 is sound with respect to this semantics. First we give a semantics for expressions using the type-free interpretation of expressions. The semantic domain $\mathbf{D}$ for the interpretation is the complete partially ordered set (*cpo*) satisfying the following domain equation (we can find such $\mathbf{D}$ in the universal domain $\mathbf{T}_\perp{}^\omega$ by the well known techniques [Plotkin 78] after appropriate encoding of truth values, natural numbers and labels/tags, and we usually omit the isomorphisms between $\mathbf{D}$ and the right-hand sum cpo). For details on cpos, we follow [Plotkin 83] and [Barendregt 81].

$$v \in \mathbf{D} \cong \mathbf{A}_0 \oplus \mathbf{A}_1 \oplus \mathbf{F} \oplus \mathbf{R} \oplus \mathbf{U} \oplus \mathbf{W}$$

where

- $\mathbf{A}_0 = \mathbf{T}_\perp$, the flat (pointed) cpo of truth values;
- $\mathbf{A}_1 = \mathbf{N}_\perp$, the flat (pointed) cpo of natural numbers;
- $f \in \mathbf{F} = [\mathbf{D} \to \mathbf{D}]$ is for function values;
- $q \in \mathbf{R} = [\mathbf{Label}_\perp \to_\perp \mathbf{D}]$ is for record values;
- $u \in \mathbf{U} = [\mathbf{Label}_\perp \times \mathbf{D}]$ is for variant (tagged union) values;
- $\mathbf{W} \stackrel{\text{def}}{=} \{?\}_\perp$

  where

  ? is the value modeling run-time type errors

  and we write its image as *wrong*, i.e. $wrong \stackrel{\text{def}}{=} in_{\mathbf{W}}(?)$;
- $\oplus$ indicates the coalesced sum construction of cpos;
- $\to$ is the domain constructor of function space.
- $\to_\perp$ is the domain constructor of strict function space.

We also need a few auxiliary domains for environments:

$\varepsilon \in \mathbf{Env} = \mathbf{EEnv} \times \mathbf{IEnv}$	the domain of environments;
$\zeta \in \mathbf{EEnv} = \mathbf{Var}_\perp \to_\perp \mathbf{D}$	the domain of valuations for ordinary variables;
$\xi \in \mathbf{IEnv} = \mathbf{IVar}_\perp \to_\perp \mathbf{D}$	the domain of valuations for implementation variables.

We interpret each expression of μFinal via its erasure, in other words, we give μFinala type-free interpretation. The semantic equations for expressions are shown in Fig. 8 (here we assume a semantic function $\mathcal{K}_i$ for each base type ι_i for the interpretation of its constants) where $in_{\mathbf{X}}$, $out_{\mathbf{X}}$ and $is_{\mathbf{X}}$ are usual primitive operations for sum domains.

$$\mathcal{E} : \mathbf{Exp} \to (\mathbf{Env} \to \mathbf{D})$$

$$\begin{aligned}
\mathcal{E}[\![x]\!]\varepsilon &= \textit{let } \langle \zeta, \xi \rangle = \varepsilon \textit{ in } \zeta[\![x]\!] \textit{ end};\\
\mathcal{E}[\![r]\!]\varepsilon &= \textit{let } \langle \zeta, \xi \rangle = \varepsilon \textit{ in } \xi[\![r]\!] \textit{ end};\\
\mathcal{E}[\![c_{ij}]\!]\varepsilon &= in_{\mathbf{A}_i}(\mathcal{K}_i[\![c_{ij}]\!]);\\
\mathcal{E}[\![\lambda x{:}\sigma.e]\!]\varepsilon &= \textit{let } \langle \zeta, \xi \rangle = \varepsilon \textit{ in } in_{\mathbf{F}}(\lambda v \in \mathbf{D}.\mathcal{E}[\![e]\!]\langle \zeta[x \mapsto v], \xi \rangle) \textit{ end};\\
\mathcal{E}[\![ee']\!]\varepsilon &= \textit{if } is_{\mathbf{F}}(\mathcal{E}[\![e]\!]\varepsilon) \textit{ then } out_{\mathbf{F}}(\mathcal{E}[\![e]\!]\varepsilon)(\mathcal{E}[\![e']\!]\varepsilon) \textit{ else } wrong;\\
\mathcal{E}[\![\{l_1 = e_1, \ldots, l_n = e_n\}]\!]\varepsilon &= in_{\mathbf{R}}(\lambda l \in \mathbf{Label}_\perp.\textit{if } l = l_1 \textit{ then } \mathcal{E}[\![e_1]\!]\varepsilon\\
&\qquad \textit{elseif } \ldots\\
&\qquad \textit{elseif } l = l_n \textit{ then } \mathcal{E}[\![e_n]\!]\varepsilon\\
&\qquad \textit{else } wrong);\\
\mathcal{E}[\![e.l]\!]\varepsilon &= \textit{if } is_{\mathbf{R}}(\mathcal{E}[\![e]\!]\varepsilon) \textit{ then } out_{\mathbf{R}}(\mathcal{E}[\![e]\!]\varepsilon)(l) \textit{ else } wrong;\\
\mathcal{E}[\![[l = e]]\!]\varepsilon &= in_{\mathbf{U}}\langle l, \mathcal{E}[\![e]\!]\varepsilon \rangle;\\
\mathcal{E}[\![\textbf{case } e \textbf{ of } l_1 \textbf{ then } e_1, \ldots, l_n \textbf{ then } e_n]\!]\varepsilon &= \textit{if } is_{\mathbf{U}}(\mathcal{E}[\![e]\!]\varepsilon) \textit{ then}\\
&\qquad \textit{let } \langle l, v \rangle = out_{\mathbf{U}}(\mathcal{E}[\![e]\!]\varepsilon) \textit{ in}\\
&\qquad\quad \textit{if } l = l_1 \textit{ then}\\
&\qquad\qquad \textit{if } is_{\mathbf{F}}(\mathcal{E}[\![e_1]\!]\varepsilon) \textit{ then } out_{\mathbf{F}}(\mathcal{E}[\![e_1]\!]\varepsilon)(v) \textit{ else } wrong\\
&\qquad\quad \textit{elseif } \ldots\\
&\qquad\quad \textit{elseif } l = l_n \textit{ then}\\
&\qquad\qquad \textit{if } is_{\mathbf{F}}(\mathcal{E}[\![e_n]\!]\varepsilon) \textit{ then } out_{\mathbf{F}}(\mathcal{E}[\![e_n]\!]\varepsilon)(v) \textit{ else } wrong\\
&\qquad\quad \textit{else } wrong\\
&\qquad \textit{end}\\
&\qquad \textit{else } wrong;\\
\mathcal{E}[\![\textbf{fix}(e)]\!]\varepsilon &= \textit{if } is_{\mathbf{F}}(\mathcal{E}[\![e]\!]\varepsilon) \textit{ then let } f = out_{\mathbf{F}}(\mathcal{E}[\![e]\!]\varepsilon) \textit{ in } \bigsqcup_n f^n(\perp_{\mathbf{D}}) \textit{ end}\\
&\qquad \textit{else } wrong.
\end{aligned}$$

Figure 8. The Semantic Equations for Expressions of μFinal.

Next we give a semantics for types based on a kind of partial equivalence relation models.

Definition 16. Let X be a set.

(1) A *partial equivalence relation* (*per* for short) on X is a symmetric and transitive binary relation on X.

(2) Let P be a per on X. Then define the *domain* of P, $|P|$, by:

$$|P| \stackrel{\text{def}}{=} \{v \in X \mid \langle v, v \rangle \in P\}.$$

(3) Let P and Q be pers on X. Then define the function space per, $P \to Q$, by:

$$\langle f, g \rangle \in P \to Q \iff \forall v, v' \in X.[\langle v, v' \rangle \in P \Rightarrow \langle f(v), g(v') \rangle \in Q].$$

(4) Let P and Q be pers on X. Then define the product per, $P \times Q$, by:

$$\langle \langle v, w \rangle, \langle v', w' \rangle \rangle \in P \times Q \iff \langle v, v' \rangle \in P \text{ and } \langle w, w' \rangle \in Q.$$

(5) Let P be a per on X and $x \in |P|$. Then define

$$[x]_P \stackrel{\text{def}}{=} \{y \in X \mid \langle x, y \rangle \in P\}.$$

(6) Let P be a per on X and $S \subseteq X$. Then define the *restriction of P on S*, $P\lceil S$, by:

$$P\lceil S \stackrel{\text{def}}{=} \{\langle u, v \rangle \in P \mid u \in S \text{ and } v \in S\}.$$

In order to interpret types as pers on **D**, we require the domain of each per corresponding to a type to be a sub-cpo of **D**.

Definition 17.

(1) Let P be a per on the cpo **D**. Then P is *complete* iff P satisfies both of the following conditions:
   (a) $\langle \bot_{\mathbf{D}}, \bot_{\mathbf{D}} \rangle \in P$; and
   (b) P is closed under lubs of ω-chains, i.e.,

$$\forall i \in \omega . \langle v_i, w_i \rangle \in R \implies \langle \bigsqcup_{i \in \omega} v_i, \bigsqcup_{i \in \omega} w_i \rangle \in P.$$

(2) **CPER** denotes the collection of complete pers (*cpers* for short) on **D**.

It is easily shown that $\langle \mathbf{CPER}, \subseteq \rangle$ is a complete lattice and has greatest lower bounds as intersections. Note that least upper bounds in **CPER** are not simple unions in general.

The semantic equations for types are as follows:

$$\mathcal{T} : \mathbf{Type} \to \mathbf{CPER}$$

$$\begin{aligned}
\mathcal{T}[\![Bool]\!] &= \{\langle d, d \rangle \mid d \in \mathbf{B}_\bot\}; \\
\mathcal{T}[\![Int]\!] &= \{\langle d, d \rangle \mid d \in \mathbf{N}_\bot\}; \\
\mathcal{T}[\![\sigma_1 \to \sigma_2]\!] &= \mathcal{T}[\![\sigma_1]\!] \to \mathcal{T}[\![\sigma_2]\!]; \\
\mathcal{T}[\![\{l_1 : \sigma_1, \ldots, l_n : \sigma_n\}]\!] &= \bigcap_{i=1}^{n} \{\langle q, q' \rangle \mid q, q' \in \mathbf{R} \text{ and } \langle q(l_i), q'(l_i) \rangle \in \mathcal{T}[\![\sigma_i]\!]\}; \\
\mathcal{T}[\![[l_1 : \sigma_1, \ldots, l_n : \sigma_n]]\!] &= \bigcup_{i=1}^{n} \{\langle \langle l_i, v \rangle, \langle l_i, v' \rangle \rangle \mid \langle l_i, v \rangle, \langle l_i, v' \rangle \in \mathbf{U} \text{ and } \langle v, v' \rangle \in \mathcal{T}[\![\sigma_i]\!]\} \\
\mathcal{T}[\![\rho r : \tau . \{\phi_1, \ldots, \phi_k\}]\!] &= \textit{let } R = \mathcal{T}[\![\tau]\!] \\
&\quad \textit{and } S = \{v \in \mathbf{D} \mid \bigwedge_{j=1}^{k} \mathcal{A}[\![\phi_j]\!]\langle \bot_{\mathbf{EEnv}}, [r \mapsto v] \rangle\} \\
&\quad \textit{in } R\lceil S \textit{ end}.
\end{aligned}$$

Figure 9. The Semantic Equations for Types of μFinal.

We interpret each assertion as an element of non-pointed **T**, since an assertion must be always either *true* or *false* even if evaluation of some of the expressions contained in it would not terminate.

$$\mathcal{A} : \mathbf{Assertion} \to (\mathbf{Env} \to \mathbf{T})$$

$$\begin{aligned}
&\mathcal{A}[\![e_1 \leqslant e_2 : \sigma]\!]\varepsilon = (\mathcal{E}[\![e_1]\!]\varepsilon \sqsubseteq \mathcal{E}[\![e_2]\!]\varepsilon); \\
&\mathcal{A}[\![\textbf{forall } x : \sigma . \phi]\!]\varepsilon = \textit{let } \langle \zeta, \xi \rangle = \varepsilon \textit{ in } \forall v \in |\mathcal{T}[\![\sigma]\!]| . \mathcal{A}[\![\phi]\!]\langle \zeta[x \mapsto v], \xi \rangle \textit{ end}.
\end{aligned}$$

Figure 10. The Semantic Equations for Assertions of μFinal.

We must check the well-definedness of the semantic function $\mathcal{T}$. For this purpose we need a lemma. Its proof shows why we imposed syntactical constraint (a) to μFinal in Section 3.

Lemma 18.

(1) Let e be closed except for free occurrences of r and at least one of the occurrences of r be strict in e. Then

$$\mathcal{E}[\![e]\!]\langle \zeta, \xi[r \mapsto \bot_{\mathbf{D}}] \rangle = \bot_{\mathbf{D}}.$$

(2) Let ϕ be a well-formed closed assertion except for free occurrences of r. Then

$$\mathcal{A}[\![\phi]\!]\langle \zeta, \xi[r \mapsto \bot_{\mathbf{D}}] \rangle = \mathit{true}.$$

Proof. (1) By induction on the structure of e.

(2) By (1) and the syntactical constraint (a) in Section 3, the denotation of the left-hand subexpression of the atomic assertion in ϕ is $\bot$, hence the statement holds. ∎

Lemma 19.
(1) Let $P \in \mathbf{CPER}$ and $S \subseteq \mathbf{D}$ be pointed and closed under lubs of ω-chains. Then $P\lceil S \in \mathbf{CPER}$.
(2) Let $P, Q \in \mathbf{CPER}$. Then $P \times Q, P \to Q \in \mathbf{CPER}$.

Proof. (1) Simple calculation.

(2) Proved by Amadio in [Amadio 91], §1.4 (1). ∎

Theorem 20 Well-definedness of $\mathcal{T}$. *$\mathcal{T}$ is well-defined. That is, for each $\tau \in \mathbf{Type}$,*
(1) $\mathcal{T}[\![\tau]\!] \in \mathbf{CPER}$;
(2) $\langle wrong, wrong\rangle \notin \mathcal{T}[\![\tau]\!]$.

Proof. (1) As in [Cardone 91], it can be shown that each semantic clause in Fig. 9 except for the algebraic-type one is well-defined and that $\mathcal{T}$ preserves completeness, hence we only have to show the well-definedness of the algebraic-type clause. We show this fact by induction on the nesting level of algebraic types. Following is the induction step. (The base case is obvious.)

Let $\rho r{:}\tau.\{\phi_1, \ldots, \phi_k\}$ be algebraic and suppose $\mathcal{T}[\![\tau]\!]$ is complete, then we only have to show that the set

$$S \stackrel{\text{def}}{=} \{v \in \mathbf{D} \mid \bigwedge_{j=1}^{k} \mathcal{A}[\![\phi_j]\!]\langle \bot_{\mathbf{EEnv}}, [r \mapsto v]\rangle\}$$

is pointed and closed under the lubs of ω-chains, then the well-definedness of algebraic-type clause follows by Lemma 19.

Pointedness: By Lemma 18 (2), $\bot_{\mathbf{D}} \in S$.

Closedness under lubs: Suppose for all $i \in \omega$, $v_i \in S$. Then for each $1 \le j \le k$,

$$\mathcal{A}[\![\phi_j]\!]\langle \bot_{\mathbf{EEnv}}, [r \mapsto v_i]\rangle = \mathit{true},$$

hence we have to show

$$\mathcal{A}[\![\phi_j]\!]\langle \bot_{\mathbf{EEnv}}, [r \mapsto \bigsqcup_{i\in\omega} v_i]\rangle = \mathit{true}. \tag{a}$$

Suppose $\phi_j \equiv \forall x_{j1}{:}\sigma_{j1}. \ldots .\forall x_{jm_j}{:}\sigma_{jm_j}.e_{\mathrm{L}j} \leqslant e_{\mathrm{R}j} : \tau_j.$ and define

$$f_{\mathrm{L}} \stackrel{\text{def}}{=} \lambda v'_1, \ldots, v'_{m_j}, v \in \mathbf{D}.\mathcal{E}[\![e_{\mathrm{L}j}]\!]\langle [x_{j1} \mapsto v'_1, \ldots, x_{jm_j} \mapsto v'_{m_j}], [r \mapsto v]\rangle,$$
$$f_{\mathrm{R}} \stackrel{\text{def}}{=} \lambda v'_1, \ldots, v'_{m_j}, v \in \mathbf{D}.\mathcal{E}[\![e_{\mathrm{R}j}]\!]\langle [x_{j1} \mapsto v'_1, \ldots, x_{jm_j} \mapsto v'_{m_j}], [r \mapsto v]\rangle,$$

then f_{L} and f_{R} are continuous with each argument. Hence, for each $i \in \omega$,

$$\begin{aligned}
&\mathcal{A}[\![\phi_j]\!]\langle \bot_{\mathbf{EEnv}}, [r \mapsto v_i]\rangle \\
=&\forall v'_1 \in |\mathcal{T}[\![\sigma_{j1}]\!]|. \ldots .\forall v'_{m_j} \in |\mathcal{T}[\![\sigma_{jm_j}]\!]|.\mathcal{A}[\![e_{\mathrm{L}j} \leqslant e_{\mathrm{R}j} : \tau_j]\!]\langle [x_{j1} \mapsto v'_1, \ldots, x_{jm_j} \mapsto v'_{m_j}], [r \mapsto v_i]\rangle \\
=&\forall v'_1 \in |\mathcal{T}[\![\sigma_{j1}]\!]|. \ldots .\forall v'_{m_j} \in |\mathcal{T}[\![\sigma_{jm_j}]\!]|.[f_{\mathrm{L}}(v'_1)\ldots(v'_{m_j})(v_i) \sqsubseteq f_{\mathrm{R}}(v'_1)\ldots(v'_{m_j})(v_i)]
\end{aligned}$$

Therefore, by the continuity of f_{L} and f_{R},

$$\forall v'_1 \in |\mathcal{T}[\![\sigma_{j1}]\!]|. \ldots .\forall v'_{m_j} \in |\mathcal{T}[\![\sigma_{jm_j}]\!]|.[f_{\mathrm{L}}(v'_1)\ldots(v'_{m_j})(\bigsqcup_{i\in\omega} v_i) \sqsubseteq f_{\mathrm{R}}(v'_1)\ldots(v'_{m_j})(\bigsqcup_{i\in\omega} v_i)] = \mathit{true}$$

so (a) holds.

(2) By induction on the structure of types. ∎

Intuitively speaking, our notion of type is a collection of values satisfying *at least* some particular properties (the set of operation actable to the value, constraints on the value which can be specified by a set of inequations).

Hence it is natural to request that, if each value of an ω-chain satisfies such properties, then the supreme of the chain must also satisfy those properties. This corresponds to the completeness condition requested to our pers (the pointedness is necessary since we want to have **fix** on all types).

On the other hand, when $v_1 \sqsubseteq v_2$, v_1 has less information than v_2 does, so v_1 may not satisfy some of the properties that v_2 does. This is the reason why we have not requested the downward closedness like in ideals [Cardelli 84] nor the closedness under approximations like in the class of pers used by Amadio and Cardone [Amadio 91, Cardone 91].

This abandonment of the approximation-closedness forces us to discard the inverse limit construction making the semantic domain $\mathbf{D}$ and we have obtained it in the universal domain $\mathbf{T}_\perp{}^\omega$, since pers on $\mathbf{D}_\infty$ is naturally requested to be closed under approximations from the construction of elements of $\mathbf{D}_\infty$. It is guaranteed by Theorem 11 in [Plotkin 78] that our semantic domain, $\mathbf{D}$, can be obtained as a retract of $\mathbf{T}_\perp{}^\omega$.

One drawback of our semantics is that the computation of a functional application cannot be performed within a type in general. To be more concrete, let f be a function from type σ to τ and a be a value of σ, then $f(a)$ must be calculated using bases $\langle e_i \rangle_{i \in \omega}$ of a. The point is that some of these bases may not belong to the sub-cpo (the domain of a per) corresponding to the type of a, σ, hence we must perform this calculation in the whole domain $\mathbf{D}$. This is the cost we have paid for our more expressive type system.

We now turn to the soundness of our type theory μFINAL with respect to this semantics.

Definition 21. An environment $\varepsilon = \langle \zeta, \mu \rangle$ is said to *respect bases* Γ, Δ (notation: $\varepsilon \models \Gamma, \Delta$) iff it satisfies both of the following two conditions:

(1) $\zeta \models \Gamma$, i.e., for any variable $x \in \mathrm{dom}(\Gamma)$, $\zeta[\![x]\!] \in |\mathcal{T}[\![\Gamma(x)]\!]|$; and
(2) $\xi \models \Delta$, i.e., for any implementation variable $r \in \mathrm{dom}(\Delta)$, $\xi[\![r]\!] \in |\mathcal{T}[\![\Delta(r)]\!]|$.

Finally, we show that the theory μFINAL is sound with respect to this semantics. First, we give some definitions and a lemma.

Definition 22.

(1) Let Σ be a sentence of μFINAL. Then Σ is *satisfied* under an environment $\varepsilon = \langle \zeta, \mu \rangle$ (notation: $\varepsilon \models \Sigma$) iff either one of the following cases holds:
   (a) when $\Sigma \equiv \sigma <: \tau$,
   $$\mathcal{T}[\![\sigma]\!] \subseteq \mathcal{T}[\![\tau]\!];$$
   (b) when $\Sigma \equiv \Gamma, \Delta \rhd e : \sigma$,
   $$\varepsilon \models \Gamma, \Delta \implies \mathcal{E}[\![e]\!]\varepsilon \in |\mathcal{T}[\![\sigma]\!]|;$$
   (c) when $\Sigma \equiv \Gamma, \Delta \rhd \phi$,
   $$\varepsilon \models \Gamma, \Delta \implies \mathcal{A}[\![\phi]\!]\varepsilon = \mathit{true}.$$

(2) Let Σ be a sentence of μFINAL. Then Σ is *valid* (notation: $\models \Sigma$) iff $\varepsilon \models \Sigma$ for any environment $\varepsilon \in \mathbf{Env}$.

Lemma 23 Substitutivity Lemma.

(1) If $\vdash \Gamma[x : \sigma'], \Delta \rhd e : \sigma$ *and* $\vdash \Gamma, \Delta \rhd e' : \sigma'$, *then for all* $\langle \zeta, \xi \rangle \in \mathbf{Env}$ *such that* $\langle \zeta, \xi \rangle \models \Gamma, \Delta$,

$$\mathcal{E}[\![e[x := e']]\!]\langle \zeta, \xi \rangle = \mathcal{E}[\![e]\!]\langle \zeta[x \mapsto \mathcal{E}[\![e']\!]\langle \zeta, \xi \rangle], \xi \rangle.$$

(2) If $\vdash \Gamma, \Delta[r : \sigma'] \rhd e : \sigma$ *and* $\vdash \Gamma, \Delta \rhd e' : \sigma'$, *then for all* $\langle \zeta, \xi \rangle \in \mathbf{Env}$ *such that* $\langle \zeta, \xi \rangle \models \Gamma, \Delta$,

$$\mathcal{E}[\![e[r := e']]\!]\langle \zeta, \xi \rangle = \mathcal{E}[\![e]\!]\langle \zeta, \xi[r \mapsto \mathcal{E}[\![e']\!]\langle \zeta, \xi \rangle] \rangle.$$

Proof. (1), (2) By induction on the structure of e. ∎

Now we can state and prove the soundness theorem for μFINAL.

Theorem 24 Soundness Theorem. *The theory* μFINAL *is sound with respect to this semantics; i.e., for any sentence* Σ *of* μFINAL,

$$\vdash \Sigma \implies \models \Sigma.$$

Proof. By induction on the structure of Σ using Lemma 23. ∎

Versions of this theorem have been presented in forms for special cases (cf. [Cardelli 84]):

Corollary 25 Semantical Soundness Theorem. *If an expression is syntactically typable, then it does not cause any run-time type error. That is,*

$$\vdash \Gamma, \Delta \rhd e : \sigma \quad \Longrightarrow \quad \forall \varepsilon \models \Gamma, \Delta. [\mathcal{E}[\![e]\!]\varepsilon \in |\mathcal{T}[\![\sigma]\!]|].$$

In other words,

$$\vdash \Gamma, \Delta \rhd e : \sigma \quad \Longrightarrow \quad \forall \varepsilon \models \Gamma, \Delta. [\mathcal{E}[\![e]\!]\varepsilon \neq \mathit{wrong}]. \ \blacksquare$$

Corollary 26 Semantical Subtyping Theorem. *Let σ and τ be types of* μFinal. *Then*

$$\sigma <: \tau \quad \Longrightarrow \quad \mathcal{T}[\![\sigma]\!] \subseteq \mathcal{T}[\![\tau]\!]. \ \blacksquare$$

6. Directions of Future Research and Related Works

The motivation of our work originates from the editors' Foreword of the proceedings of International Workshop of Semantics of Data Types [KMP 84]. It says that *"The Symposium was intended to bring these somewhat disparate groups together with a view to promoting a common language ...,"* but unfortunately there have been hardly any efforts to integrate logical and algebraic approaches to abstract data types by now. What we have shown in this paper is that the type system with inequational assertions is a natural extension of a typed λ-calculus with record types and the complete partial equivalence relation model is rich enough to interpret types with inequational assertions.

Our system can be called a type system combining programming types (usual types of μFun) and specification (inequational assertions as partial correctness requirements) used to write specifications for *verification* as well as *executable* programs, hence our μFinal is a good candidate for foundations of type systems of *functional wide-spectrum languages* such as Extended ML [Sannella and Tarlecki 89]. The present work is just the first step toward the goal of incorporating algebraic structures with logical types with domain-theoretical foundations. There remain many interesting issues as follows:

(1) to give our language typed interpretations;

(2) to strengthen our inequality "$\leqslant$" to "=" in assertions;

(3) to extend our system to second-order calculi with polymorphism, existentially quantified types, bounded quantifications, and parameterization of types;

(4) to enrich our system with recursion on types;

(5) to incorporate more sophisticated record calculi such as row variables and selective field updating.

For (1), we have given semantics of μFinal via the erasure interpretation as shown above. In other words, we have given semantics of μFinal as a wide-spectrum functional language, but we have not constructed any typed model of μFinal as an enriched simply typed λ-calculus. We expect that our semantics will be able to be converted to a typed model as far as first-order calculi using some structures like Lindenbaum algebras without much difficulties, but as we will discuss later, the non-computable aspects of our types may have some affects in model constructions when we extend our calculus to second-order calculi such as polymorphism, ... etc.

For (2), roughly speaking, our inequality "$\leqslant$" corresponds to partial correctness in program verification while the equality "=" to total correctness, since "$e_1 \leqslant e_2$" intuitively means that, if the computation of e_1 terminates, then it must give the same result as that of e_2, but the computation e_1 may diverge on its way, while the equality requests that the results of computations of both sides must always coincide. Using "=" introduces the new problem that denotations of types may be empty. This difficulty can be overcome if we introduce the notion of *admissible types* whose assertions have no inconsistencies. Adopting this notion forces us to admit that the syntactic well-formedness of types is no longer context-free (nor even decidable). On the other hand, if the denotation of a type (with equational assertions) is not empty, then it is a cper, hence this approach serves enough interests.

For (3), the type system with existentially quantified types and algebraic types will give the full modeling of ADTs. For example, we can define the abstract data type of the group-like structure as:

$$\begin{aligned}\textbf{type}\ \mathit{Group} = \exists G.\rho a\mathit{GroupOp}\{&(\_\cdot\_): G \to G \to G,\\ &(\_)^{-1}: G \to G,\\ &e: G\\ &|\textbf{forall}\ x, y, z: G.x \cdot (y \cdot z) = (x \cdot y) \cdot z : G,\\ &\textbf{forall}\ x: G.x \cdot (x)^{-1} = e : G,\\ &\textbf{forall}\ x: G.e \cdot x \leqslant x : G\}.\end{aligned}$$

But the only known model of existentially quantified types is based on intervals of ideals [Martini 88] and Cardone has pointed out that the profinite pers (cpers closed under approximations) cannot be used to model such types. On the other hand, the extension making the system a polymorphic type system seems possible as long as we adopt the erasure interpretation, since cpers are closed under arbitrary intersection. With these extensions, we can obtain *true* abstract type constructors. Furthermore, bounded quantification of types is a very interesting extension for this type system. For example, if we can write $\forall\tau \leq \mathit{StackOpImpl}. \ldots$, then this bounded quantification means "for all types τ *having at least the stack structure* ..." and such description will be very useful for specifications of modular programming [Cardelli and Wegner 85].

For (4), it is worth noting that our semantic function $\mathcal{A}$ is not continuous, hence types of our system are not computable in general. The computability of types are essential, however, for this extension. To overcome this difficulty, we may substitute the object-level equality operator "eq_σ" for the present "$\sqsubseteq$" (or "$=$") of the meta-level in the semantic function $\mathcal{A}$ and simultaneously replace non-pointed $\mathbf{T}$ by the usual pointed domain $\mathbf{T}_\perp$. But This also involves several problems. First, not every type has it own computable equality operator, hence, we must characterize the class of types equipped with such equality. This class corresponds to *eqtype* in Standard ML, and its domain-theoretical characterization is unknown hence may be interesting. Second, this approach introduces the problem of empty types again. Third, the inverse image of $\{\mathit{true}\}$ is not pointed. If we avoid this problem by considering the inverse image of $\{\perp, \mathit{true}\}$, then the corresponding proof system loses the transitivity rule.

For (5), record calculi with raw variables and selective updating are proposed to be very useful in modeling of inheritances of object-oriented programming [Cardelli and Mitchell 91] and incorporating these ideas with our algebraic types must give a good foundation of functional object-oriented wide-spectrum languages.

On foundations of wide-spectrum languages, the Martin-Löf type system [Martin-Löf 84] is also such a system [NPS 90]. But it has several drawbacks: the first is that it does not support fixed-points and limits only total functions losing some computable total functions. The second is rather pragmatic problem, writing a specification of an ADT with his type system using equality types for assertions, means that the execution of the extracted program for the ADT contains a construction of the proof of "this ADT is correctly implemented," which is intuitively irrelevant for the execution of the *intended* program. The Göteborg group has introduced their Subset Theory [NPS 90] to remedy this inefficiency, but it does not preserve De Bruijn-Curry-Howard correspondence, while the original system does and it is the main merit of the Martin-Löf system.

Relating our results to T-algebras is another interesting theme. For strict and continous $f \in \mathbf{D} \to \mathbf{D}$ and continuous $g \in \mathbf{D} \to \mathbf{D}$, predicates of the form $P(v) \equiv f(v) \sqsubseteq g(v)$ are ω-inductive, and such predicates has a strong connection with T-algebras (cf. [Plotkin 83], Chapter 5, Theorem 4). [Lehmann and Smyth 81] used T-algebras to interpret types with operations in domain theory. But as Pierce has pointed out in [Pierce 91, p. 41], *"this construction works only for algebras without equations. The framework has apparently never been extended to include algebras with equations."*

Finally, we summarize the related works. In [Cardelli 84], Cardelli used the ideal models to interpret types, whose downward-closedness condition cannot be satisfied by algebraic types. [Cardone 91] gave a typed interpretation of the second-order typed λ-calculus with fixed-points, records and variants using more restrictive pers than we have used in this paper. [Amadio 91] interpreted types via realizability over the reflexive domain and studied basic properties of several kinds of pers. Actually, we used some of his results in this paper. In [Abadi and Plotkin 90], they studied the closedness condition of pers, especially with respect to Plotkin powerdomain construction. In all of these studies, the pers considered are those over $\mathbf{D}_\infty$ and must be closed under approximations. This condition is not satisfied by interpretations of assertions, however,

hence we have used a kind of universal domain satisfying a retraction instead of $\mathbf{D}_\infty$ and complete pers over it for interpretations. [Bruce and Longo 88] presented a per model for the bounded polymorphic calculus without recursion either on expressions or on types. The most interesting work on the semantics of bounded second-order λ-calculi is by Martini [Martini 88], who successfully gives a semantics to the second-order calculus with bounded existential types, bounded polymorphic types and fixed-points on expressions using intervals of ideals as interpretations of types. But none of these previous works have considered *algebraic structures*. Actually, Mitchell and Plotkin have already pointed out the necessity for considering structures, but the approach they have suggested is based on the formulas-as-types correspondence and contains the same problems as noted about Martin-Löf's system.

Acknowledgements

The author wish to express his deepest thanks to Professor Henk Barendregt for his invaluable advice on an earlier version of this paper and for his warm encouragement. He also strongly wish to express his gratitude for a referee of TLCA '93 who pointed out an essential problem in the earlier version of this paper and gave the author very constructive comments and encouragement. Furthermore, the author thanks Yugo Kashiwagi for his enthusiastic discussions and valuable suggestions. Dr. Aart Middeldorp's *true* encouragements much increased the author's motivation to complete this work. The author is obliged to Kenroku Nogi for his encouragement and comments. Last but not least, the author is grateful to Dr. Eiichi Maruyama, the former general manager of Hitachi Advanced Research Laboratory, and Nobuyoshi Domen, the general manager of Hitachi Systems Development Laboratory, who gave me the chance to start this work, and Dr. Shojiro Asai, the general manager of Hitachi Advanced Research Laboratory, for providing the ideal research environment for the author to continue his work.

References

[Abadi and Plotkin 90] Abadi, M. and G. D. Plotkin: A Per Model of Polymorphism and Recursive Types, *5th IEEE Conf. of Logic in Computer Science*, 355–365 (1990).

[Amadio 91] Amadio, R. M.: Recursion over Realizability Structures, *Inform. Comput.* **91**, 55–85 (1991).

[Barendregt 81] Barendregt, H. P.: *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, Amsterdam (1981).

[Bruce 92] Bruce, K. B.: *A Paradigmatic Object-Oriented Programming Languages: Design, Static Typing and Semantics*, Technical Report CS-92-01, Williams College (Jan. 31, 1992).

[Bruce and Longo 88] Bruce, K. B. and G. Longo: A Modest Model of Records, Inheritance and Bounded Quantification, *3rd IEEE Conf. of Logic in Computer Science*, 38–50 (1988); a revised version appeared in *Inform. Comput.* **87**, 196–240 (1990).

[Bruce and Mitchell 92] Bruce, K. and J. C. Mitchell: PER Models of Subtyping, Recursive Types and Higher-order Polymorphism, *16th ACM Symp. on Principles of Programming Languages*, 316–327 (1992).

[Cardelli 84] Cardelli, L.: A Semantics of Multiple Inheritances, in [KMP 84], 51–67; a revised version appeared in *Inform. Comput.* **76**, 138–164 (1988).

[Cardelli and Mitchell 91] Cardelli, L. and J. C. Mitchell: Operations on Records, *Math. Struct. Comput. Sci.* **1**, 3–48 (1991).

[Cardelli and Wegner 85] Cardelli, L. and P. Wegner: On Understanding Types, Data Abstraction, and Polymorphism, *ACM Comput. Surv.* **17** (1985).

[Cardone 91] Cardone, F.: Recursive Types for Fun, *Theoret. Comput. Sci.* **83**, 29–56 (1991).

[Ehrig and Mahr 85] Ehrig, H. and B. Mahr: *Fundamentals of Algebraic Specification 1*, Springer-Verlag, Berlin (1985).

[KMP 84] Kahn, G., D. B. MacQueen, and G. Plotkin (eds.): *Semantics of Data Types*, Proceedings of International Symposium, Sophia-Antipolis, June 1984, Lecture Notes in Computer Science **173**, Springer-Verlag, Berlin (1984).

[Lehmann and Smyth 81] Lehmann, D. J. and M. B. Smyth: Algebraic Specification of Data Types: A Synthetic Approach, *Math. Syst. Theory* **14**, 97–139 (1981).

[Martini 88] Martini, S.: Bounded Quantification Have Interval Models, *1988 ACM Conf. on LISP and Functional Programming*, 164–173 (1988).

[Martin-Löf 84] Martin-Löf, P.: *Intuitionistic Type Theory*, Bibliopolis, Napoli (1984).

[Mitchell and Plotkin 85] Mitchell, J. C. and G. D. Plotkin: Abstract Types Have Existential Type, *12th ACM Symp. on Principles of Programming Languages*, 37–51; a revised version appeared in *ACM Trans. Prog. Lang. Syst.* **10**, 470-502 (1988).

[MTH 90] Milner, R. et al.: *The Definition of Standard ML*, MIT Press, Cambridge MA (1990).

[NPS 90] Nordström, B. et al.: *Programming in Martin-Löf's Type Theory*, Clarendon Press, Oxford (1990).
[Pierce 91] Pierce, B. C.: *Basic Category Theory for Computer Scientists*, MIT Press, Cambridge MA (1991).
[Plotkin 77] Plotkin, G. D.: LCF Considered as a Programming Language, *Theoret. Comput. Sci.* **5**, 223–255 (1977).
[Plotkin 78] Plotkin, G.: $\mathbf{T}^{\omega}$ as a Universal Domain, *J. Comput. Syst. Sci.* **17**, 209–236 (1978).
[Plotkin 83] Plotkin, G.D.: *Domains*, Advanced Postgraduate Course Notes, Department of Computer Science, University of Edinburgh (1983).
[Reynolds 83] Reynolds, J. C.: Types, Abstraction and Parametric Polymorphism, *Information Processing 83* (R. E. A. Mason ed.), 513–523, North-Holland, Amsterdam (1983).
[Reynolds 85] Reynolds, J. C.: Three Approaches to Type Structures, *TAPSOFT-CAAP '85* (H. Ehrig et al. eds.), Lecture Notes in Computer Science **185**, 97–138, Springer-Verlag, Berlin (1985).
[Sannella and Tarlecki 89] Sannella D. and A. Tarlecki: *Toward Formal Development of ML Programs: Foundations and Methodology — Preliminary Version*, Technical Report ECS-LFCS-89-71, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh (1989).
[Thompson 91] Thompson, S.: *Type Theory and Funcional Programming*, Addison-Wesley, Reading MA (1991).